Build Your Own Wizards

Doug Hennig

Wizards provide an ideal way of holding your users hands through complex, multi-step tasks. This month's Reusable Tools column presents a set of classes you can use to build your own wizards.

We've all used wizards for several years now. FoxPro has included wizards since version 2.x (the Setup Wizard was one of the first), Windows 95 includes all kinds of wizards for things like setting up printers and dial-up connections, and even end-user applications like Word and Publisher include wizards. Why all the emphasis on wizards?

A wizard is the ideal user interface for:

- tasks that have a lot of steps
- complex tasks where it's easy to make a mistake or forget a step
- a user base that's somewhat inexperienced with computers

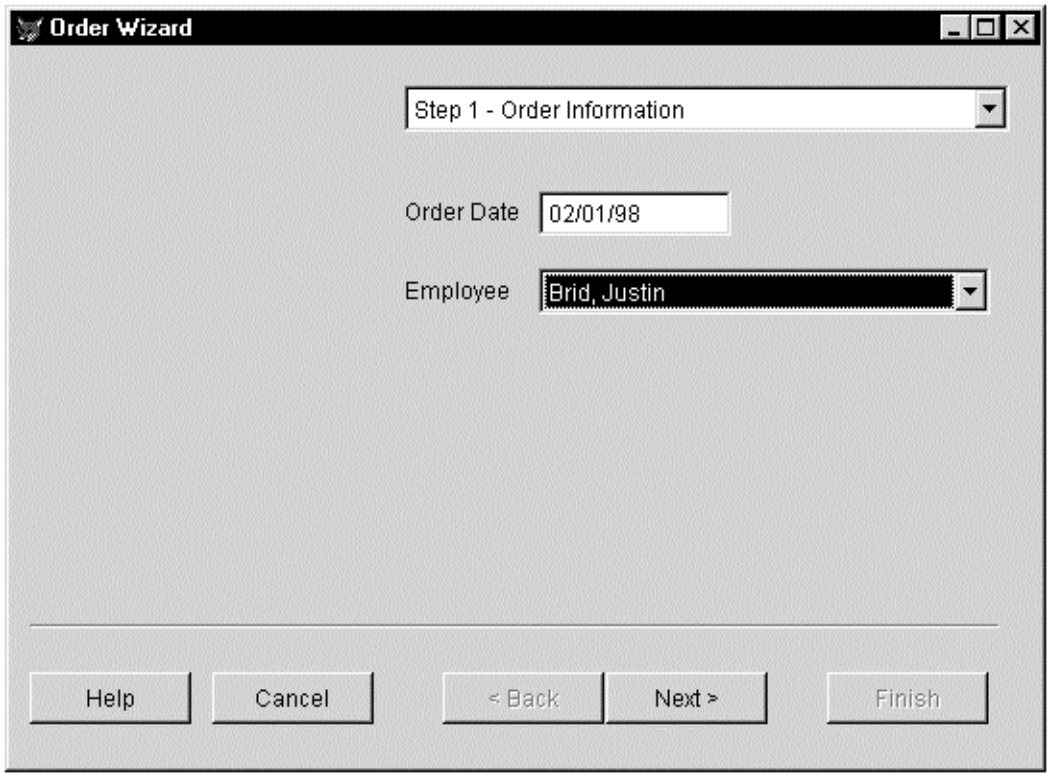
This is because a wizard takes an overall task and divides it into a set of logical steps. The user can choose the order in which to perform the steps or even return to an earlier step to correct an error. It provides a way to hand-hold the user through an operation, making it easy for the user to understand while at the same time ensuring the task is properly completed.

This doesn't mean you'll replace all your data entry forms with wizards. Forms, even those with page frames and lots of controls, are still often the best user interface for viewing and editing records in tables. However, consider using a wizard when you have a clearly defined sequence of steps that make up a task. Here are some examples of where a wizard might be appropriate:

- An Export Wizard can be a generic tool you include in your applications to allow the users to get data out in a format other than reports. Step 1 might be to select the table to export from, Step 2 to select the fields to export, and Step 3 to choose the format (text file, Excel, Word, email, HTML, etc.) and file name. The Finish step would do the actual exporting.
- An Import Wizard might be complicated to build, but would be a handy way for users to get existing data into your application. GoldMine (a Windows contact manager we use at Stonefield) has a slick import function, although it's not exactly laid out like a wizard in the version we use. You specify the data source type (for example, DBF or text file) and file name, then define a mapping between the fields in the data source and the contact tables. You can either view data source field names or the data content (you can even move to the next and previous records, which may help identify the data). Mappings can be saved in case you regularly import the same file structures.
- I'm currently working on an application for a client in which several tasks don't easily fit into the concept of forms. For example, to schedule a photo shoot, the user has to identify which magazine and issue the shoot is for, select the items to be shot, enter details on how to shoot them, and print several reports; all this is considered to be a single task. The Photo Shoot Wizard will ensure the user doesn't skip any steps in this complex task.

Although they do vastly different things, wizards have a common interface. They have a series of steps for the user to perform, each step having its own set of instructions and controls in the same form (usually done as a page in a tabless page frame). Back and Next buttons take the user from one step to the next, and a combo box allows the user to choose any step in any order (assuming the wizard's rules permit this). A Cancel button cancels the task and the Finish button completes it. Sounds like the place for a class definition, doesn't it? Figure 1 shows a wizard we'll build in this article, the Order Wizard.

Figure 1. The Order Wizard.



Wizard Form

SFWizardForm (defined in SFWIZARD.VCX) is the base class for wizard forms. It's subclassed from SFForm, our Form base class defined in SFCTRLS.VCX. It has a private datasession so it doesn't interfere with anything else and optimistic buffering so changes made to tables can easily be reverted.

Since a wizard's tasks typically consist of several steps, the form has an SFWizardPageFrame object (which we'll look at later) with one page per step. To maintain the same interface as Microsoft's wizards, the page frame has its Tabs property set to .F. so the user can't select a step by clicking on a tab; they have to select a step from a combo box or using Next and Back buttons. Also, all controls on the page frame appear on the right half of each page, leaving room for graphics representing each step on the left half.

SFWizardForm has the custom public properties shown in Table 1.

Table 1. Custom properties of SFWizardForm.

Property	Description
aSteps	Array of steps; one row per step and three columns: 1 = step displayed to the user 2 = "step complete" rule expression 3 = .T. if the step is complete
nCurrentStep	Current step
nMaxSteps	Number of steps in the wizard

The aSteps array drives a couple of things: a combo box (cboSteps) from which the user can select a step (its RowSource property is set to Thisform.aSteps), and determining when to enable the Next and Finish buttons. Since the user can't go on to a particular step until all tasks up to that step are complete, the second column of aSteps contains an expression for each step that, if evaluated to .T., indicates the step is done. The expression could check for certain controls on that step's page having valid values or it could call a custom method for more complex validation. The third column of the array is set to .T. if this step and all preceding steps are done; only then can the user move to the next or last step.

The Init method dimensions the aSteps array to as many rows as there are steps (the nMaxSteps property) and three columns, then calls the SetupSteps method to populate this array. It selects the first step

by calling the SelectStep method and passing it 1 for the step number. It then calls the RefreshSteps method to refresh all controls based on the selected step. Here's the code for Init:

```
with This
  assert .nMaxSteps > 0 ;
    message 'Set nMaxSteps to a valid value'
  assert .pgfWizard.PageCount = .nMaxSteps ;
    message "nMaxSteps doesn't match pgfWizard.PageCount"
  dimension .aSteps[.nMaxSteps, 3]
  .SetupSteps()
  .SelectStep(1)
  .RefreshSteps()
  dodefault()
endwith
```

SetupSteps is an abstract method (you fill in the code that defines the contents of the aSteps array for the wizard you're creating) although it has a single line to requery the cboStep combobox (which is bound to aSteps); at the end of the custom code, you just use DODEFAULT() to update the combobox. Here's an example of what might go in this method (taken from the Order Wizard):

```
with This
  .aSteps[1, 1] = 'Step 1 - Order Information'
  .aSteps[1, 2] = 'not empty(ORDERS.ORDER_DATE) ' + ;
    'and not empty(ORDERS.EMP_ID)'
  .aSteps[2, 1] = 'Step 2 - Select Customer'
  .aSteps[2, 2] = 'not empty(ORDERS.CUST_ID)'
  .aSteps[3, 1] = 'Step 3 - Select Products'
  .aSteps[3, 2] = 'ORDERS.ORDER_AMT <> 0'
  .aSteps[4, 1] = 'Step 4 - Finish'
  .aSteps[4, 2] = 'not empty(ORDERS.SHIP_VIA)'
  dodefault()
endwith
```

The expression that goes in the second column can test the value of controls on the step's page if desired. To make it easier to reference them, the page is automatically referenced via a WITH statement before trying to evaluate the expression. This means you can use something like "not empty(.txtDate.Value)" rather than having to specify the complete object hierarchy (such as "not empty(Thisform.pgfWizard.Page3.txtDate.Value)". Note this code doesn't populate the third column of aSteps; that column is taken care of by other methods.

The SelectStep method is called to select a particular step; Init calls it to select step 1, the Next and Back buttons call it to select the next or previous step, and the cboSteps combo box calls it to select the specified step. If SelectStep returns .F., the specified step cannot be selected because prior steps aren't finished. SelectStep calls the abstract method StepDone, which allows you to perform any code prior to leaving a step or prevent the step from being left under certain conditions (returning .F. does this). After setting the nCurrentStep property to the specified step, the abstract StepSelected method is called, where you can put any code that needs to occur when a page is selected.

```
lparameters tnStep
local llReturn
with This
  assert between(tnStep, 1, .nMaxSteps) ;
    message 'Invalid step passed to SelectStep'
  if (tnStep <= .nCurrentStep or ;
    .aSteps[tnStep - 1, 3]) and ;
    .StepDone(.nCurrentStep)
    .nCurrentStep = tnStep
    .StepSelected(.nCurrentStep)
    .Refresh()
    llReturn = .T.
  else
    llReturn = .F.
  endif tnStep <= .nCurrentStep ...
endwith
return llReturn
```

The RefreshSteps method is called when the value of any control changes (we'll see how that's done later). It evaluates the second column of each row in the aSteps array, and sets the third column to .T. only if each step and all prior ones are complete. If a step can't be selected because prior steps aren't complete, RefreshSteps adds a backslash prefix to the step description in the first column of aSteps so it'll be disabled in the cboSteps combo box.

```
local llOK, ;
  lnSteps, ;
  lnI, ;
  lcPrompt, ;
  lcRule, ;
  loControl
with This
  if not empty(.aSteps[1, 1])
    llOK    = .T.
    lnSteps = alen(.aSteps, 1) + 1
    for lnI = 2 to lnSteps
      lcPrompt = iif(lnI = lnSteps, '', ;
        .aSteps[lnI, 1])
      lcRule   = .aSteps[lnI - 1, 2]
      with .pgfWizard.Pages[lnI - 1]
        llOK = llOK and evaluate(lcRule)
      endwith
      .aSteps[lnI - 1, 3] = llOK
      do case
        case lnI = lnSteps
        case left(lcPrompt, 1) = '\' and llOK
          .aSteps[lnI, 1] = substr(lcPrompt, 2)
        case left(lcPrompt, 1) <> '\' and not llOK
          .aSteps[lnI, 1] = '\' + lcPrompt
      endcase
    next lnI
```

After testing each step, RefreshSteps refresh all controls that need refreshing. It calls the RefreshSteps method of the page frame (which we'll look at later) and the Refresh method of any control that has an lRefreshSteps property set to .T.

```
    for each loControl in .Controls
      do case
        case lower(loControl.Class) = 'sfwizardpageframe'
          loControl.RefreshSteps()
        case type('loControl.lRefreshSteps') = 'L' and ;
          loControl.lRefreshSteps
          loControl.Refresh()
      endcase
    next loControl
  endif not empty(.aSteps[1, 1])
endwith
```

The Click methods of the Back and Next buttons call the SelectStep method with the previous or next step number (Thisform.nCurrentStep - 1 or Thisform.nCurrentStep + 1). The Back button is only enabled when the user is on step 2 or later, and the Next button is only enabled if the user isn't on the last step and the current step is completed (Thisform.aSteps[Thisform.nCurrentStep, 3] is .T.).

The Finish button is only enabled when the last step is done (Thisform.aSteps[Thisform.nMaxSteps, 3] is .T.). Its Click method calls the Finish method of the wizard form; this is an abstract method (except it releases the form in this class) because the action to take when the user clicks on this button varies from wizard to wizard.

The Cancel method of the form is called from the Click method of the Cancel button and from the QueryUnload method of the form (so clicking on the form's Close box cancels the wizard). This method reverts all changes in all tables, so you may not need to customize this method in a particular wizard. Here's the code:

```
local laCursors[1], ;
  lnI, ;
  lcCursor
for lnI = 1 to aused(laCursors)
  lcCursor = laCursors[lnI, 1]
  if cursorgetprop('Buffering', lcCursor) > 1
    tablerevert(.T., lcCursor)
  endif cursorgetprop('Buffering', lcCursor) > 1
next lnI
```

The form's KeyPreview property is set to .T. so it can process keystrokes before any other control. The reason for this is so the form's KeyPress method can treat PgUp as if the user clicked the Back button and PgDn as if the user clicked Next. Here's the code:

```
LPARAMETERS nKeyCode, nShiftAltCtrl
with This
  do case
    case nKeyCode = 18 and .cmdBack.Enabled
      .SelectStep(.nCurrentStep - 1)
    case nKeyCode = 3 and .cmdNext.Enabled
      .SelectStep(.nCurrentStep + 1)
  endcase
endwith
```

Wizard Controls

SFWIZARD.VCX contains several classes that can be used as controls in wizard forms. They're subclasses of the appropriate class in SFCTRLS.VCX.

SFWizardPageFrame, which is used for the page frame in SFWizardForm, has a RefreshSteps method used to refresh only those controls that need refreshing; it's called from the RefreshSteps method of the form. It has the following code:

```
local loPage, ;
  loControl
for each loPage in This.Pages
  for each loControl in loPage.Controls
    if type('loControl.lRefreshSteps') = 'L' and ;
      loControl.lRefreshSteps
      loControl.Refresh()
    endif type('loControl.lRefreshSteps') = 'L' ...
  next loControl
next loPage
```

The Init method makes the page frame the same size as the form and the Refresh method sets This.ActivePage to Thisform.nCurrentStep so the page for the current step is selected.

SFWizardCommandButton has an lRefreshSteps property, which defaults to .T. and indicates whether the object should be refreshed when the wizard form's RefreshSteps method is called. All buttons on SFWizardForm are SFWizardCommandButton buttons; this way, the Next, Back, and Finish buttons can be refreshed as soon as the user changes something.

SFWizardCheckBox, SFWizardComboBox, SFWizardOptionGroup, SFWizardSpinner, and SFWizardTextBox also have an lRefreshSteps property and all have the following code in their InteractiveChange and ProgrammaticChange methods:

```
local lnPos, ;
  lcAlias, ;
  lcField
with This
  lnPos = at('.', .ControlSource)
  if .lRefreshSteps and lnPos > 0 and ;
    not .Value == evaluate(.ControlSource)
    lcAlias = left(.ControlSource, lnPos - 1)
    lcField = substr(.ControlSource, lnPos + 1)
    replace (lcField) with .Value in (lcAlias)
  endif .lRefreshSteps ...
  .AnyChange()
```

```
endwith
Thisform.RefreshSteps()
```

This code ensures the Value of the control is written to its ControlSource (if there is one) before calling any other method; failing to do so may cause step completion expressions that test the value of a field to fail, since the field's value hasn't been updated from the control yet. The code then calls the custom AnyChange method (which our base classes have for code needed when a control's Value changes) and the form's RefreshSteps method. As a result, whenever any change (programmatic or interactive) is made to a control, the form and all necessary controls are immediately refreshed, enabling or disabling the buttons and cboSteps combo box as necessary.

Building a Wizard

OK, now that we've discussed the components of a wizard, let's actually build one (if I were Ken Levy, I'd create a wizard for building wizards, the Wizard Wizard, but I digress). The ORDERWIZ form accompanying the source code for this article is an Order Wizard. This wizard makes it easy for users to enter an order for products sold to a customer. The data files for this wizard come from the VFP TESTDATA database, so copy the contents of the SAMPLES\DATA subdirectory of your VFP directory to the same directory you install this source code in before running this form.

The Init method of ORDERWIZ adds a new record to the ORDERS table, defaulting the order date to today's date and the order number to one more than the highest existing order number. It then uses DODEFAULT() to do the rest of the SFWizardForm behavior.

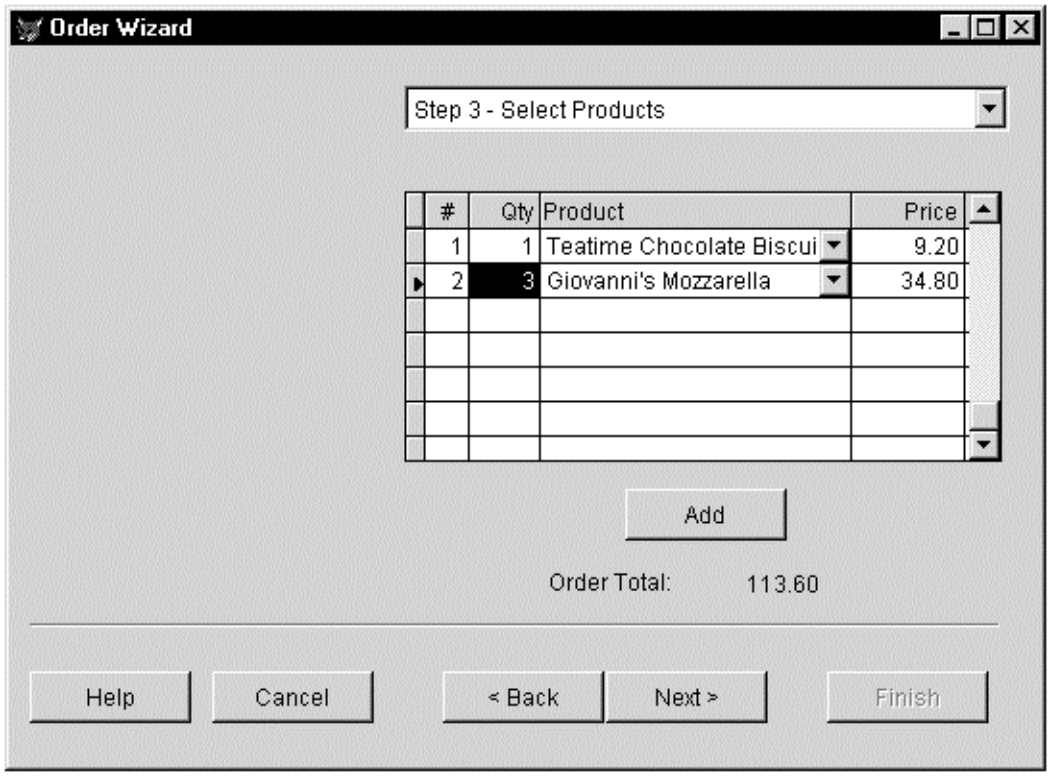
The nMaxSteps property is set to 4 and the PageCount property of the page frame are set to 4 because that's how many steps there are in our wizard. The aSteps array is populated with the information about these steps in the SetupSteps method (we looked at the code for this method earlier).

Step 1 is for order information, so Page 1 of the page frame contains a textbox for the order date (bound to ORDERS.ORDER_DATE) and a combo box for the employee who filled the order (populated with names from the EMPLOYEE table and bound to ORDERS.EMP_ID). This step isn't complete until the user enters values in both controls.

Step 2 is to select the customer who placed the order, so Page 2 of the page frame contains a combo box populated with names from the CUSTOMER table and bound to ORDERS.CUST_ID, and text boxes bound to address fields from the CUSTOMER table. The AnyChange method of this combo box positions the CUSTOMER table to the selected customer's record so the customer text boxes will show the address for the selected customer. These text boxes are read-only so the address can't be changed in this form, but it would be a simple change to give the wizard that capability. This step isn't complete until a customer has been chosen. The form's StepDone method, which is automatically called when the user leaves a step, has code that populates some fields in the ORDERS table from the current CUSTOMER record when the user leaves step 2.

The user selects the products ordered in Step 3 (shown in Figure 2). Page 3 of the page frame contains a grid bound to the ORDITEMS table, an Add command button, and a textbox showing the order total. Clicking on the Add button adds a new record to the ORDITEMS table, and sets ORDITEMS.LINE_NO to the next line number for this order. The first column of the grid is bound to ORDITEMS.LINE_NO and is read-only. The second column is bound to ORDITEMS.QUANTITY. The third column, which is bound to ORDITEMS.PRODUCT_ID, contains a combo box populated with product names from the PRODUCTS table, making it easy to select a product. Column 4 is bound to ORDITEMS.UNIT_PRICE, which is set by the AnyChange method of the product combo box to the unit price of the selected product. The custom UpdateTotal method of the form is called when the user changes the quantity, product, or unit price; this method calculates and displays the order total. Step 3 is complete once the order total is non-zero. The form's StepDone method has code that eliminates partially entered records in ORDITEMS (for example, if the user clicks on the Add button but doesn't enter the quantity or product for the new line).

Figure 2. Step 3 in the Order Wizard.



In Step 4, the user enters the shipping method (using a combo box showing shippers) and freight amount. This step is complete once a shipper has been chosen, and since this is the last step, the Finish button is enabled. The Finish method of the form uses TABLEUPDATE() to write out all changes in all tables.

Conclusion

One improvement that could be made to SFWizardForm is to make it data-driven. A Wizards table could contain the information that currently must be coded into the SetupSteps method of each wizard: the text to display for each step and the expression that when evaluated to .T. indicates the step is complete. If you used such a table, the SetupSteps method of SFWizardForm would then open the table, grab all the records for the current wizard, and populate the aSteps array (probably using a SQL SELECT statement).

Whether data-driven or not, the SFWizardForm and accompanying classes in SFWIZARD.VCX allow you to easily add wizards to applications. I expect to be using these classes more and more in the applications I develop.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America, spoke at the 1997 Microsoft FoxPro Developers Conference (DevCon), and will be speaking at the 1998 DevCon. He is a Microsoft Most Valuable Professional (MVP). CompuServe 75156,2326 or dhennig@stonefield.com.